The EASE Actor Development Environment

Paul Scerri and Nancy E. Reed

Technical Report: TACSIM-99-01
Real-time Systems Laboratory
Department of Computer and Information Science
Linköping University, S-581 83 Linköping, Sweden
pausc,nanre@ida.liu.se

October 1999

Abstract

In interactive simulations it is often desirable to have intelligent actors playing the roles of humans. Drawing on a wide range of previous work this paper presents a system that is intended to reduce some of the difficulties involved in the development of actors. We present a system called EASE (End-user Actor Specification Environment) that provides tools and methods to support end user development of intelligent actors. The tools support the whole development process from design to testing. The EASE actor architecture is a multi-agent system where a process of contract making and negotiation between agents determines the actions of the actor.

1 Introduction

In modern, complex, interactive simulations it is often highly desirable to have intelligent actors playing the roles of humans. The actors' task is difficult – sensing the (simulated) environment, choosing a course of action that flexibly and intelligently follows designer intentions and sending appropriate commands back to the environment, all in real-time. The actors' reasoning may need to be very complex taking into account a variety of factors including the current situation, a variety of concurrent, potentially conflicting goals, team members, opponents, previous actions, resource constraints and so on [Tambe *et al.*, 1995b].

In order for the simulation in which the actors are embedded to be useful the actors must usually act in a human-like manner. Often knowledge of precisely how an actor should act will be *expert knowledge* hence it is desirable to have *domain experts*, as opposed to actor experts, specifying actor behavior.

A plethora of recent work has resulted in architectures for actors with a wide range of abilities. However in many cases the actor architectures are difficult to use, support only an adhoc development process and provide little support for

reuse. Virtually no architecture allows domain experts to directly specify actor behavior. This in turn leads to an expensive, frustrating development process [Jennings *et al.*, 1998, Nwana, 1999, Wooldrige and Jennings, 1998]. It is clearly desirable to improve the process of creating intelligent actors.

Drawing on a wide range of previous work this paper presents a system that is intended to reduce some of the difficulties involved in the development of useful, complex actors. The system, called EASE (Enduser Actor Specification Environment), is a group of tools and an associated methodology for the development of complex, intelligent actors. The system provides support for all stages of development from design through testing to reuse. In particular the system is intended to represent the first step toward putting actor development capabilities into the hands of the domain experts. By providing structure and support for a simple, rapid development process along with an accessible actor architecture, EASE provides the basis for end user development.

1.1 Overview of EASE

Within EASE an actor specification consists of a hierarchy of *agents* where each agent is responsible for some aspect of the overall actor behavior (see Figure 1). Each agent takes into account only it's specific task and is hence fairly simple. Below an agent in the hierarchy are other agents that perform parts of it's behavior.

At runtime an actor's specification is turned into a multi-agent system where overall actor behavior is determined by a continuous process of contract making and negotiation between agents. Agents form a hierarchy of contracts then agents at the bottom of the hierarchy negotiate amongst themselves over the actual output of the actor.

On top of the multi-agent actor architecture EASE enforces a methodology for actor development that covers all stages of development, from design through to reuse. EASE also provides tool support for the implementation task which allows completely graphical development, achievable by non-programmers. The tools have been designed to make reuse as simple as possible, primarily by enforcing strict modularity. To make the process of testing as quick and painless as possible integrated tool support exists for quickly inspecting and debugging actors at runtime. The development aids with the EASE system combined with an underlying powerful agent runtime engine allow relatively inexperienced users to create useful actors for complex simulation environments.

1.2 Target Domain

The initial target domain for this system is simulated aircraft pilots. The TACSI air-combat simulator [Aircraft, 1995], developed at Saab AB, is used for both training of human pilots and testing of new systems. In this domain new actor behavior is often required, either to test new systems or to train pilots. It is desirable that the engineers and pilot trainers that actually use the simulator can define the behavior of the pilot actors. These people are well educated and

accustomed to using computers but not necessarily AI/actor experts. Many of the scenarios involved are very similar increasing the desirability of easily reusing parts of existing specifications. The pilot actors need to appear to be intelligent and act realistically in a very complex environment. The actuators for the actor, i.e. the aircraft controls, are extremely complex and allow many degrees of freedom [Tambe *et al.*, 1995a].

1.3 Related Work

Recently there has been a lot of interest in development methodologies for agents. Examples include methodologies for Belief Desire Intention (BDI) agents [Kinny and Georgeff, 1996], for behavior based agents [Bryson, 1998], for distributed multi-agent systems [Bussman, 1998], for safety critical multi-agent systems [Bouchefra *et al.*, 1998] and a more general method for a range of agent oriented systems [Wooldridge *et al.*, 1999].

Development environments, often including substantial graphical support, exist for building actors or agents for a variety of domains. For example the Bond system for collaborative network agents [Bölöni and Marinescu, 1999], Jackal for agent based communication infrastructure [Cost *et al.*, 1999], Zeus for distributed agents [Nwana *et al.*, 1999], the Icon Modelling Tool (IMT) for mobile agents [Falchuk and Karmouch, 1998] and MissionLab[MacKenzie, 1996] for robotics. None of these tools are suited for creating complex actors for simulation environments.

Systems do exist for developing actors for interactive simulations. The AgentSheets system, for example, is designed to allow users with very limited computing experience, often children, to develop fairly complex actors [Repenning, 1993]. AgentSheets agents, however, are greatly restricted in the type of sensing and acting they can do. KidSim is also designed to allow very inexperienced users to develop intelligent actors [Smith *et al.*, 1997]. Like Agentsheets, KidSim is very restrictive in the possible behavior and types of environment.

EASE fits into a niche between the systems mentioned above. EASE provides a structured development methodology to improve the process of building actors. However, somewhat in contrast to the methodologies above, the emphasis of this process is to build the systems *quickly* and *easily*. Like AgentSheets, EASE is not intended to be used only by actor or programming experts. However the increased abilities of EASE actors, as compared to those of AgentSheets, means that EASE users will need to be more experienced than AgentSheets users. In many ways EASE attempts to meet the same goals for simulation actor development as MissionLab does for robot programming.

2 Multi-agent Decision Making

In this section the functioning of the multi-agent system that controls an actor is explained. In the following section the development process for such a multi-agent system is described.

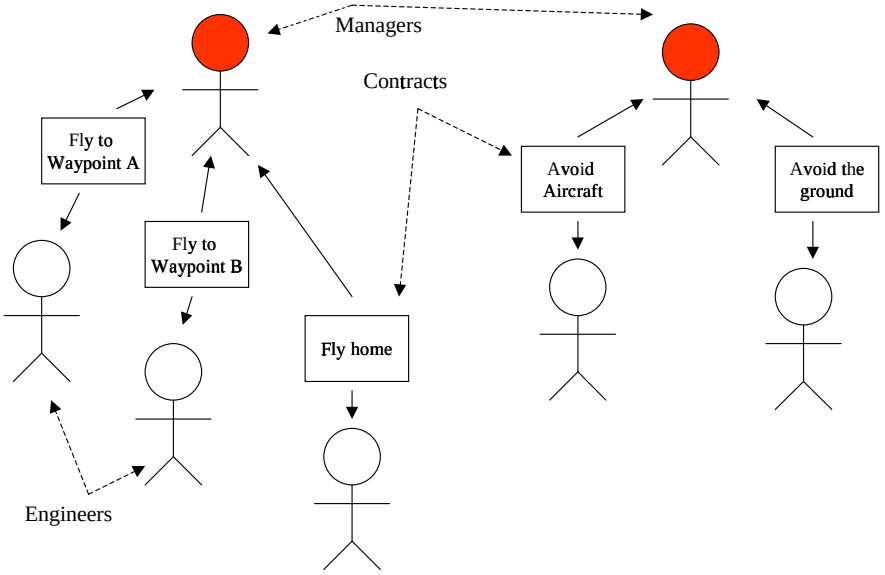

Figure 1: Part of an actor specification for a simple pilot patrol actor

There are two main types of agent, referred to as *managers* and *engineers*, in the multi-agent system of an actor. The agents are arranged hierarchically into a forest of trees. Engineers will be the leaf nodes of the trees, i.e. at the bottom of the hierarchies, while all other nodes are managers. Connections between the nodes are in the form of *contracts*.

Manager type agents are charged with more abstract tasks. To achieve their tasks manager agents *contract* other agents, either engineers or other managers, to fulfill specific parts of the task. A manager may have contracts with zero or more agents at any time and may break existing contracts or make new contracts over time. For example one manager agent may be responsible for a particular patrol mission. The manager could contract a sequence of agents one for taking off, one for flying through each of the patrol's waypoints and one for landing the aircraft.

Contracts form the only connections between managers and other agents in the system. The contracts always form the agents into a strict hierarchy. At design time a designer hardcodes the contracts a manager should make to handle a particular situation, i.e. there is no service brokering or negotiation to find an appropriate agent. An agent will not refuse a contract request from a manager for a new contract, in fact at the implementation level a new agent is created specifically to service the contract.

At runtime, if for some reason an agent is unable to fulfill its contract, either temporarily or permenantly, it will inform the manager that contracted it. The contracting manager may then choose another course of action, if it has one, or

report the failure to the manager that contracted it.

The designer will specify one or more agents to be started at runtime. The specified agents will be the only agents not contracted by another agent at runtime. Generally the starting agents will form the tops of hierarchies of agents that in turn control the actor.

A special type of manager agent, called a *list manager*, is associated with a *type of thing* in the environment. For every instance of the type sensed by the actor the list manager will contract an agent and assign it to the specific instance. For example a list manager responsible for avoiding midair collisions will contract a specific agent to avoid each particular midair obstacle. Each contracted agent needs only concern itself with the specific obstacle it was assigned.

The other main type of agent (a manager was the first) is an engineer. Engineers are at the bottom of the agent hierarchies. Engineer agents negotiate with other engineer agents over the output of the actor, i.e. the commands the actor sends to the environment. Each engineer "argues" for output values that best fulfill the task it has been contracted to achieve.

Each degree of freedom of the actor is associated with a factory, e.g. a simulated pilot may have factories for aircraft heading and speed. In this way the complex negotiations over the actors' behavior is split into a number of simpler negotiations. A factory continuously suggests possible output values to all engineers that have registered interest in its output. The interested engineers respond with their *satisfaction* with the suggested output value. At regular intervals the factory sends to the environment the output value most acceptable to the interested engineers.

For example a factory for the heading of an aircraft may have two interested engineers. One of the engineers is responsible for avoiding a particular aircraft and another is responsible for getting to a particular waypoint (see Figure 2). The *avoidance* engineer will be satisfied with any heading suggestion that leads to avoiding the other aircraft, and be more satisfied with headings that result in comfortably avoiding the aircraft. The *waypoint* agent will be satisfied with headings that result in generally heading towards the waypoint and happier with those that result in heading directly toward the waypoint. Hopefully the factory can find a heading that satisfies both the engineers.

The process of negotiation is continuous. Because the environment will be changing as the negotiation proceeds so may engineers preferences for different output values also change. When new engineers are contracted or when existing engineers are stopped the factory negotiation simply continues on, albeit with the new set of agents. The continuous nature of the negotiation means that it is an *anytime algorithm* [Zilberstein, 1996] and hence is suitable for (soft) real-time actors.

The negotiation mechanism allows multiple high level tasks to be attended to simultaneously without a designer having explicitly considered the interactions between the tasks. Each manager, responsible for one high level task, simply contracts appropriate agents to perform pieces of its overall task. At the bottom of the subsequent hierarchies the engineers associated with the different tasks

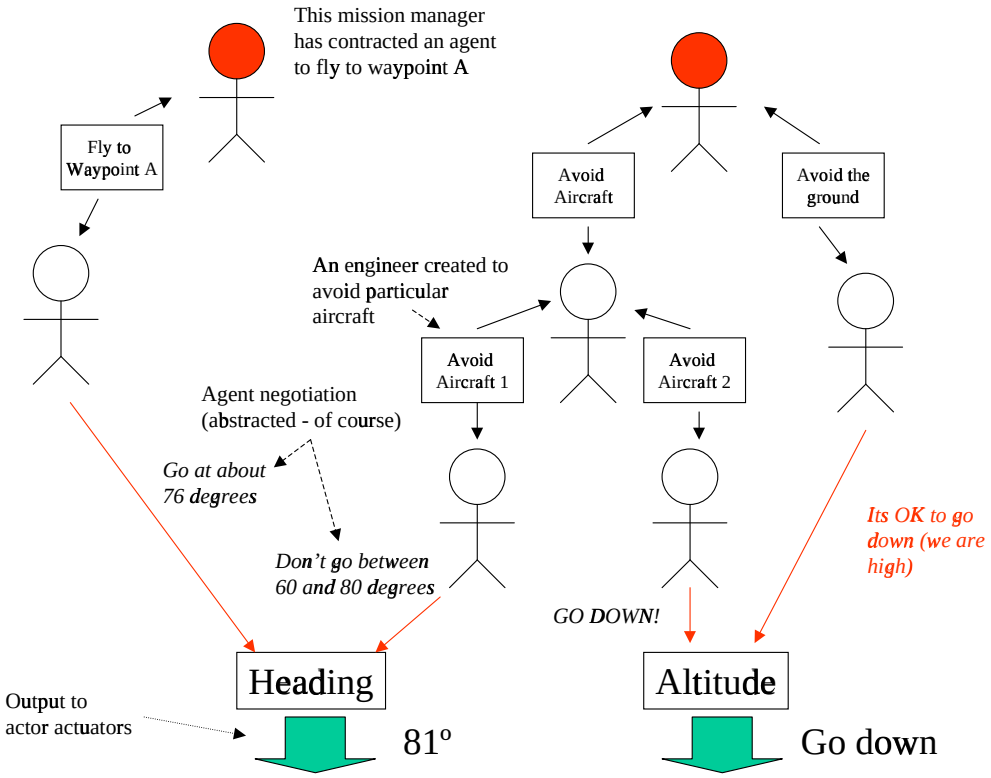

Figure 2: A snapshot of the multi-agent actor controller

negotiate amongst themselves and the factories to find a solution that achieves all, or as many as possible of the managers' tasks. The negotiation process integrates the high level tasks without designer intervention.

The earlier example, with two engineers, raises the question of what happens when the factory cannot find an output value that satisfies all of the interested agents. When not all engineers can be satisfied the *priority* of the engineers is taken into account.

The priority of an agent is a function of three factors. The first factor is the *intrinsic* priority of the agent. The intrinsic priority is a static value based on the type of task the agent has, e.g. a safety critical ground avoidance agent has a higher intrinsic priority than a fuel conservation agent. The second factor of the priority is referred to as the *organizational* priority. The organizational priority is defined by the role the agent has in the overall system. For example the aforementioned fuel conservation agent will have a high organizational priority when contracted by a manager responsible for leaving hazardous territory. The final factor of an agent's priority is referred to as *environmental* priority. The environmental priority is dynamic and varies according to the significance of the agent given the current environmental circumstances. For example a fuel conservation agent may have high environmental priority when the remaining fuel is low and the aircraft is far from base.

Although most significant for engineers, all agents have a priority. For managers the priority system, in particular the organizational factor, means that higher priority managers get more say in low level negotiations, i.e. a higher priority manager will have engineers lower down in their hierarchy with high organizational priorities and hence more say in negotiations.

3 Actor Development

EASE supports the whole development process, from design to testing to reuse. Many of the development stages are explicitly supported by tools while for other stages of development a particular methodology is advocated. In this section the development process with EASE is explained in detail with particular emphasis on the tool support provided.

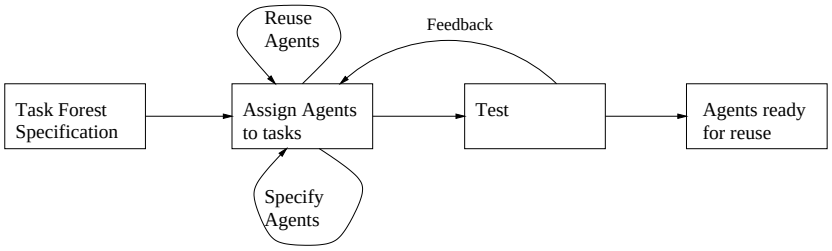

Figure 3: The actor development process

The development of an actor (see Figure 3) begins with the design of trees of tasks and behaviors that describe the overall behavior of the actor. Behaviors, continuous aspects of the actor's overall behavior, e.g. ground avoidance, are mixed freely with tasks, time bounded aspects of the actor's overall behavior, e.g. flying to a waypoint. The design need not consist of a single tree but may be a forest of trees. The tops of trees represent high level abstract behaviors of the actor. Further down the tree are more specific aspects of the actors overall behavior (see Figure 1). There is no timing or sequencing information at this stage, there is simply a breakdown of overall functionality into pieces.

The next stage of development is to assign an agent to each node in the task forest. The agent will be performing the task assigned to that node. Leaf nodes will be assigned engineers and internal nodes, managers. Branches between nodes are replaced by contract specifications. The internal behavior of the agents enforces appropriate sequencing of tasks.

The assignment of agents to nodes is done in two steps. Firstly, existing libraries of agents are consulted to find agents that can be reused. Often reusable agents would have been created for other, similar scenarios. In most cases if an agent is found to match a non-leaf node in the behavior forest a whole agent hierarchy will be found for the node's branch. Usually an agent can be imported without change, however some tuning may be required later to adjust the priorities of the imported agents to the new specification. The amount of tuning should be minimal and be required only at the top of the imported agent hierarchy.

When appropriate agents cannot be found in existing libraries they are created in the core of the EASE system – the agent specification tool (see Figure 4). To specify an agent the designer specifies a name, intrinsic priority, environmental priority function, a state machine for controlling behavior and any contracts or factory assignments the agent will have.

The name of the agent can be any text, hopefully describing the intended functionality of the agent. Next the designer needs to specify whether the agent is a manager or an engineer. The intrinsic priority is set via a slider and should reflect the designer's assessment of the importance of the functionality of the agent when considered in isolation. The environmental priority function is defined via the function specification system (described below). The function maps actor sensor readings to values representing the importance of the agent in the particular situation, e.g. sensor readings indicating proximity to the ground should map to high values for an agent responsible for ground avoidance.

The state machine for an agent provides the mechanism by which the decision making of an individual agent is defined. (Having state machines as the only decision making mechanism is a current specification system limitation, rather than an important aspect of the overall system.) The state machines are standard single level Moore state-machines. State transition conditions are defined with the function specification system. If the agent is an engineer, for each state in the state machine the engineer is assigned to negotiate with a specific factory (i.e. over a particular degree of freedom). The function specification system is used, once again, to define the function that the engineer

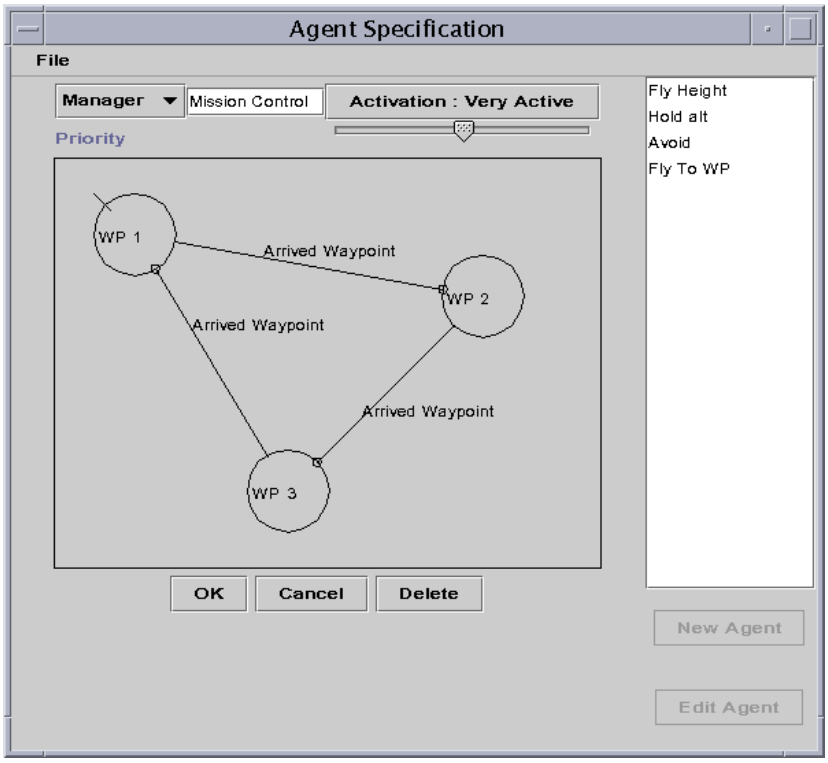

Figure 4: The Agent Specification subsystem

uses to determine the acceptability of a factory suggestion. Alternatively, if the agent is a manager, for each state the designer specifies the contracts the manager should make in each state. Specifying a contract consists of selecting which agent should be contracted and instantiating any parameters associated with the contracted agent (e.g. the waypoint to fly to).

Once an actor has been partially or fully specified it can be tested in the target simulation environment. In order to support an iterative process of testing and incrementally expanding or improving actor behavior a number of graphical interfaces display interactively and in real-time the status of the agent's reasoning system.

The main testing interface shows all the currently active agents and their current status. The interface allows the designer to see whether the right agents are being created and whether the agents seem to be reacting to the environment in the desired manner, e.g. by creating appropriate contracts of arguing for reasonable output values. Problems such as state transitions being taken at the wrong time or wrong contracts being entered into can be observed with the tool. The designer can return to the agent specification system and make appropriate changes if problems are observed.

If the behavior of the actor is not as expected, but the correct agents seem to be active and in the correct states, the designer can use another tool to view the status of negotiations (see Figure 5). This tool gives a real-time view of the output of a factory and the satisfaction of each of the interested engineers to the factories suggestions. The designer may observe that some agents, due possibly, to overly high priority are getting too much say in a negotiation. Going back to the specification the intrinsic priority of the agent (or the agent that contracted it) could be lowered or the environmental priority function for the agent (or the agent that contracted it) changed. Alternatively it may be observed that an agent seems to be calculating incorrect values. To investigate further the designer can pop up another window that allows snapshots of the details of function calculations to be displayed. If the designer finds problems here they can return to the function specification system to rectify the problem.

The cycle of specification and testing will be repeated until the required actor behavior is achieved. The final stage of the development process is making the newly created specification available for reuse. However reuse requires no further effort! New actor specifications can use parts of the current specification in a completely black box fashion.

3.1 Function Specification

The function specification system allows a designer to specify the different functions that an agent uses(see Figure 6). The requirements on the function specification system are quite imposing. Potentially extremely complex functions need to be specified, e.g. the activation function of an avoid enemy aircraft agent would take into account a wide range of factors about the relative positions of the other aircraft, in a way that does not require professional programmers and at the same time encourages reuse. The function specification system is based on

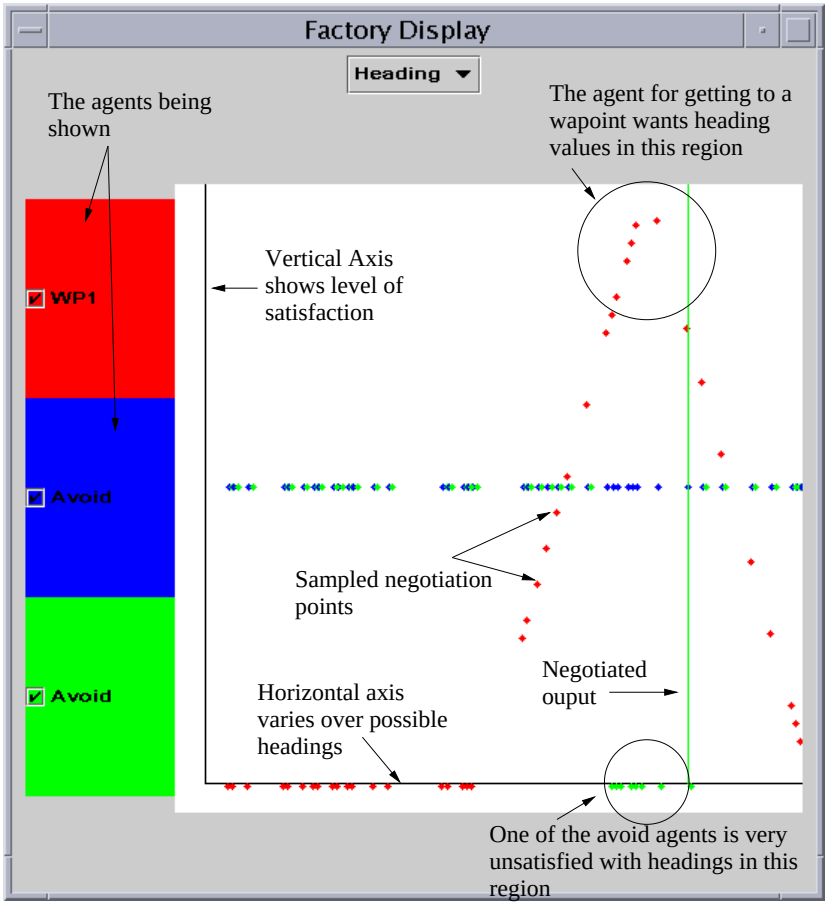

Figure 5: The Negotiation Debugger shows the current status of the negotiations.

the idea of a spreadsheet. The spreadsheet paradigm was chosen because of the successful way non-computing professionals can develop complex functionality with a spreadsheet despite not being able to program.

The function specification system does not look like a conventional spreadsheet (the cells are not laid out in grid, rather they appear in a tree structure) and has some features that do not appear in conventional spreadsheets, most importantly parameterizable cells. Because cells are parameterizable so are functions. In turn because functions are parameterizable so are agents. Parameterizable functions and agents lead to high levels of reuse, as well as reducing the specification size. The other major difference to a spreadsheet is that values and functions are not entered as text rather they are created by selecting options from lists. This removes the possibility of specification errors due to syntax or type problems.

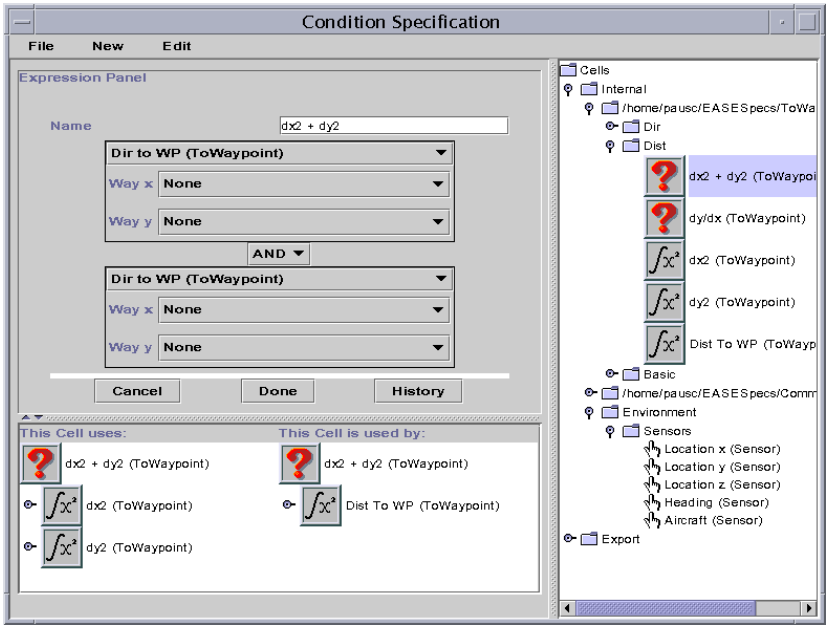

Figure 6: The Function Specification subsystem

4 Discussion

In a series of famous papers Brooks argues strongly against the dominant ways that AI practitioners went about building intelligent systems [Brooks, 1991a, Brooks, 1991b]. Rather than complex, monolithic systems incapable of interacting with the real world Brooks advocates a behavior-based approach where the overall behavior of a situated actor is broken *horizontally* into smaller pieces

of behavior. It is argued that actors should be built by creating simple behaviors then incrementally "subsuming" the existing behaviors with more complex behaviors.

Brooks' behavior-based idea is extremely promising. An intuitive breakdown of overall actor behavior and a low risk incremental development process seem immediately possible. For users unaccustomed to building actors, in particular domain experts, the behavior-based paradigm potentially offers much especially in terms of having an actor specification that matches an intuitive breakdown of the actors' task. However subsumption and behavior-based systems have not delivered all they have promised. One of the key reasons seems to be that the subtle interactions between individual behaviors means that the complexity of adding new behaviors to an actor soon becomes overwhelming [Bryson, 1998].

Recently a variety of different approaches have been taken to reduce the complexity of the interactions or, at least provide methods for allowing developers to handle the complexity better. The methods either organize behaviors differently, e.g. [Parker, 1998, Blumberg and Galyean, 1995], or combine the outputs of the behaviors in a different way, e.g. [Yen and Pfluger, 1995, Pirjanin, 1998, Riekki, 1998, Rosenblatt, 1997]. With the EASE actor architecture we are taking this trend one step further.

A behavior in a behavior-based system is an agent in EASE. By elevating behaviors to the status of agents the interactions between behaviors simplify in the same way that "agentifying" other complex systems simplifies the interactions between subsystems. In effect, using agents instead of behaviors makes behaviors "active" rather than passive entities. The interaction between agents can then be strictly controlled, through contracts and negotiations, and more easily understood – drastically reducing the complexity of the effects on overall behavior due to subtle interactions. The reduction in the amount of subtle interactions between behaviors should lead to an increase in the level of actor complexity that a designer can be reasonably expected to develop.

The EASE multi-agent system, being an extension of behavior-based ideas, shares many properties with "standard" behavior-based systems. EASE lies very much towards the reactive end of a reactive-deliberative scale. Agents are generally very simple, mapping sensor input to appropriate contracts or negotiation strategies. However this need not be the case, agents could potentially be more complex doing any type of planning or reasoning. The overall behavior of an actor is an emergent result of the agents' interactions with others and with the world.

The potentially conflicting requirements of a usable development environment and genuinely useful actors imply an underlying actor architecture with the following characteristics:

- Actor specifications with abstract, loosely coupled specification elements; and
- A runtime engine that combines the specification elements in a powerful and flexible manner.

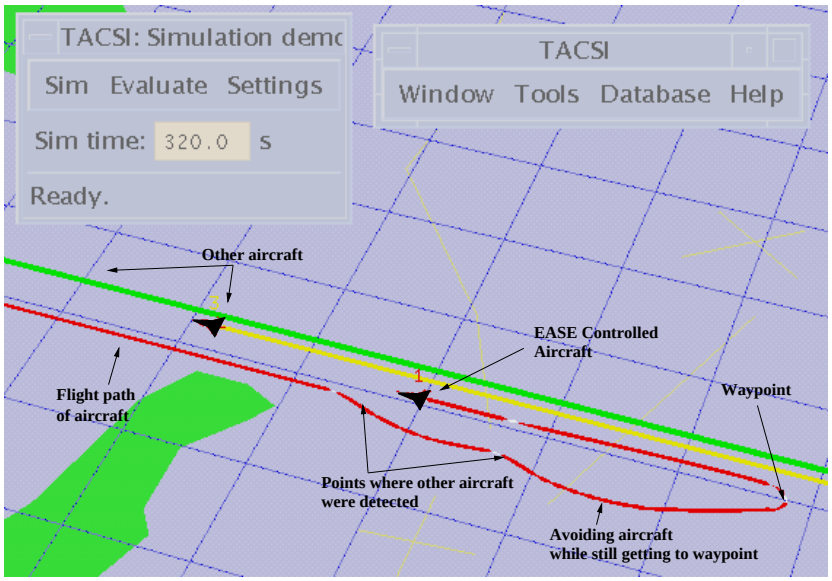

Figure 7: The result of a simulation run in TACSI.

The former characteristic addresses usability. It is likely to be difficult to build a development system that "compiles" a user specification into an actor unless there is a fairly close correspondence between the specification and runtime mechanisms (in the same way there is a close correspondence between constraint programming or imperative programming specifications and their respective runtime mechanisms.) Hence it seems reasonable to assume that the actor architecture will need to have features that support enduser development. Abstraction is a mechanism used in all kinds of specification problems to allow a designer to break problems into manageable pieces. Loose coupling between specification elements is a desirable feature of a program design the reduces complexity, makes for more readable specifications and encourages reuse.

The latter characteristic, i.e. a powerful computational engine, addresses the problem of creating genuinely useful actors. Intuitively the more flexibly and powerfully a specification can be interpreted by the runtime engine the more useful behavior will be observed from a specification of a certain size and complexity. (The same phenomena can be observed with constraint and imperative programming. The constraint runtime engine is far more powerful than an imperative one leading to smaller and simpler constraint programs for some problems.) For example, if an actor runtime engine simply chooses the first applicable situation-action rule from some list the overall behavior for even a large specification is likely to be fairly simple. At the other end of the spectrum if the runtime engine takes the specification, supplements it with common sense reasoning and previous experience then invokes sophisticated planning routines

the observed behavior of the actor will probably be fairly sophisticated. It follows that the EASE runtime engine should be as powerful as possible.

The actor architecture underlying EASE fulfills the two characteristics. The hierarchical structure and "single-mindedness" of the individual agents provides the loose coupling and abstraction desired. The multi-agent runtime engine provides the powerful reasoning system required to make useful actors.

Figure 7 shows the output from a simulation run in TACSI. There is one aircraft controlled by EASE. Notice the way the aircraft smoothly integrates getting to the waypoint and avoiding approaching aircraft. The flight path is reminiscent of obstacle avoidance paths for behavior based robots. Although this example is very simple it goes someway toward illustrating that the multi-agent system can produce some reasonable behavior. The jury is still out on whether the emergent behavior can produce usefully realistic behavior. More work needs to be done to determine precisely the strengths and weaknesses of the approach.

Both major aspects of EASE, namely the underlying computational engine and the overlying specification process, have been designed by looking at existing systems and attempting to improve *modularity*. Our previous experience developing agents suggests that modularity in agent specifications is a key to scaling up, reducing costs, improving testability and so on. The intended usage scenario for EASE makes modularity even more critical. In particular good modularity should provide the following desirable properties:

- Rapid Prototyping. Over time libraries of agents can be built up. New actors can be rapidly put together by reusing old agents, i.e. parts of old actors.
- Highly Complex Actors. As with the development and design of any complex system modularity is a key to making a specification comprehensible and manageable.
- Use by Novices. Once libraries of agents have been developed relative novices should be able to piece together agents in order to create required actors.
- Good Development Process. All reasonable software development processes rely on being able to break the problem down into pieces, developing actors should be no different.
- Development Teams. Good modularity of specification allows different developers to work on different parts of the same actor, leading to shorter development times.

Modularity was emphasized in the computational engine through limiting interactions between the agents. To an even greater extent modularity was a driving concern in the design of the development system. The modularity was largely achieved by ensuring that the tools encouraged breaking a task into pieces. The idea of specific purpose agents leaves little room for, say, mixing

flying to a waypoint with avoiding obstacles. In a complementary way the tools provide no mechanisms for considering the internals of other agents or even knowing about the existence of any agents except those directly, hierarchically related.

5 Future Work

Future work is intended to push the system even further towards the hands of domain experts rather than the present, realistic target of low level agents and functions created by agent experts and pieced together by domain experts. At the time of writing EASE was about to begin testing on site at Saab with simulation experts.

Future work on the actor architecture will include experimenting with different negotiation mechanisms, including the method the factory uses for finding new suggestion values, the protocol for the negotiation and the function the factory uses to decide which suggestion is the "best". At present negotiation only occurs at the level of engineers, in the future negotiation between managers may allow better integration of multiple high level goals.

Future work on the EASE specification system will focus on ways of encouraging designers to develop actor specifications that can be easily reused. At the other end of the development cycle the process of identifying which parts of new specifications can use existing agents and how the appropriate existing agents can be identified will be improved.

Acknowledgments

This work is supported by Saab AB, Operational Analysis division, The Swedish National Board for Industrial and Technical Development (NUTEK), under grants IK1P-97-09677 and IK1P-98-06280, and Linköping University's Center for Industrial Information Technology (CENIIT), under grant 99.7.

References

[Aircraft, 1995] Saab Military Aircraft. The TACSI users guide. technical report GDIO-MI-98:356. Technical report, Saab Military Aircraft, 1995. Edition 5.2.

[Blumberg and Galyean, 1995] Bruce Blumberg and Tinsley Galyean. Multi-level control of autonomous animated creatures for real-time virtual environments. In *Siggraph '95 Proceedings*, pages 295–304, New York, 1995. ACM Press.

[Bölöni and Marinescu, 1999] Ladislau Bölöni and Dan Marinescu. A framework for building collaborative network agents. Technical Report CSD-TR #99-001, Purdue University, 1999.

[Bouchefra *et al.*, 1998] Kamel Bouchefra, Patrick Auge, Thierry Maury, Brigitte Rozoy, and Roger Reynaud. Multi-agent based architecture specification and verification. In *Proceedings of Eleventh Workshop on Knowledge Acquisition and Modeling and Management*, 1998.

[Brooks, 1991a] Rodney Brooks. Intelligence without reason. In *Proceedings 12th International Joint Conference on AI*, pages 569–595, Sydney, Australia, 1991.

[Brooks, 1991b] Rodney Brooks. Intelligence without representation. *Artificial intelligence journal*, 47:139–159, 1991.

[Bryson, 1998] Johanna Bryson. Agent architectures as object oriented design. In Munindar Singh, editor, *The fourth international workshop on agent theories, architectures and languages (ATAL97)*. Springer Verlag, 1998.

[Bussman, 1998] S. Bussman. Agent oriented programming of manufacturing control tasks. In *Third International Conference on Multi-agent systems*, pages 57–63, Paris, 1998.

[Cost *et al.*, 1999] Scott Cost, Tim Finin, Yannis Labrou, Xiaocheng, Yun Peng, and Ian Soboroff. Agent development with Jackal. In *Proceedings of the Third Annual Conference on Autonomous Agents*, pages 358–359, Seattle, WA, May 1999.

[Falchuk and Karmouch, 1998] Benjamin Falchuk and Ahmed Karmouch. Visual modeling for agent based applications. *Computer*, 31(12):31–38, Dec 1998.

[Jennings *et al.*, 1998] Nicholas Jennings, Katia Sycara, and Michael Wooldridge. A roadmap of agent research and development. *Autonomous agents and multi-agent systems*, 1:275–306, 1998.

[Kinny and Georgeff, 1996] David Kinny and Michael Georgeff. Modelling and design of multiagent systems. In *Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages*. Lecture Notes in Computer Science, 1996.

[MacKenzie, 1996] Douglas MacKenzie. *A design methodology for the configuration of behavior-based mobile robots*. PhD thesis, Georgia Institute of Technology, 1996.

[Nwana *et al.*, 1999] Hyacinth Nwana, Divine Ndumu, Lyndon Lee, and Jaron Collis. ZEUS: A toolkit and approach for building distributed multi-agent systems. In *Proceedings of the Third Annual Conference on Autonomous Agents*, pages 360–361, Seattle, WA, May 1999.

[Nwana, 1999] Hyacinth Nwana. A perspective on software agents research. *Knowledege Engineering Review*, 1999.

[Parker, 1998] Lynne E. Parker. Alliance: An architecture for fault tolerant multi-robot cooperation. *IEEE Transactions on Robotics and Automation*, 14(2), 1998.

[Pirjanin, 1998] Paolo Pirjanin. *Multiple objective action selection and behavior fusion voting.* PhD thesis, Department of Medical Informatics and Image Analysis, Aalborg university, 1998.

[Repenning, 1993] Alexander Repenning. *AGENTSHEETS: A tool for building domain-oriented dynamic, visual environments.* PhD thesis, University of Colorado, Boulder, 1993.

[Riekki, 1998] Jukka Riekki. *Reactive task execution of a mobile robot.* PhD thesis, Infotech Oulu and Department of Electrical Engineering, University of Oulu, Oulu, Finland., 1998.

[Rosenblatt, 1997] Julio Rosenblatt. Behavior-based planning for intelligent autonomous vehicles. In *AV Symposium on Intelligent Autonomous Vehicles*, Madrid, Spain, 1997.

[Smith *et al.*, 1997] David Smith, Allen Cypher, Jim Spohrer, Apple Labs, and Apple Computer. *Software Agents*, chapter KidSim: Programming Agents without a Programming Language. AAAI Press/The MIT Press, 1997.

[Tambe *et al.*, 1995a] M. Tambe, K. Schwamb, and P. Rosenbloom. Constraints and design choices in building intelligent pilots for simulated aircraft. In *AAAI Spring symposium on "Lessons Learned from implemented software architectures for phyiscal agents"*, 1995.

[Tambe *et al.*, 1995b] Milind Tambe, W. Lewis Johnson, Randolph Jones, Frank Koss, John Laird, Paul Rosenbloom, and Karl Schwamb. Intelligent agents for interactive simulation environments. *AI Magazine*, 16(1):15–39, Spring 1995.

[Wooldridge *et al.*, 1999] Michael Wooldridge, Nicholas Jennings, and David Kinny. A methodology for agent-oriented analysis and design. In *Proceedings of the third annual conference on Autonomous agents*, pages 69–76, Seattle, WA, 1999.

[Wooldrige and Jennings, 1998] Michael Wooldrige and Nicholas Jennings. Pitfalls of agent oriented development. In *Proceedings of the Second International Conference on Autonomous Agents*, 1998.

[Yen and Pfluger, 1995] John Yen and Nathan Pfluger. A fuzzy logic based extension to Payton and Rosenblatt's command fusion method for mobile robot navigation. *IEEE Transactions on Systems, Man and Cybernetics*, 25(6), 1995.

[Zilberstein, 1996] S. Zilberstein. Using anytime algorithms in intelligent systems. *AI Magazine*, 1996.